Tabla de contenido

Manual de Ejercicio-SQL de Cero a Experto

Video 1- Introducción a Base de Datos

Para esta sección se diseñó un conjunto de ejercicios integradores seccionados por capítulos para que el alumno pueda practicar lo aprendido, pero a su vez que el proceso de resolución del problema sea un poco más retador a comparación de la sesión teórica.

Para lograr el objetivo se seleccionó una base de datos clásica para el proceso de enseñanza de MySQL.

La base de datos Northwind fue creada originalmente por Microsoft como una base de datos de ejemplo para Microsoft Access y luego también para SQL Server. Su propósito era servir como modelo de una pequeña empresa de importación y exportación que vende productos alimenticios y gourmet.

¿Qué simula Northwind? Una empresa ficticia llamada Northwind Traders, que:

- Vende productos (alimentos, bebidas, condimentos…).
- Tiene clientes en distintos países.
- Registra pedidos hechos por esos clientes
- Cuenta con empleados que gestionan pedidos.
- Tiene proveedores, categorías, detalles de cada pedido, etc.

*¿Qué contiene?*Aquí están algunas de las tablas más representativas:

Tabla	Contenido
Customers	Datos de clientes (nombre, país, teléfono, etc.)
Employees	Empleados de la empresa (nombre, cargo, jefe, etc.)
Orders	Pedidos realizados por clientes
OrderDetails	Detalles de cada pedido (producto, cantidad, precio)
Products	Productos en venta
Suppliers	Proveedores que surten productos
Categories	Categorías de los productos
Shippers	Empresas que hacen los envíos

NOTA: Tiene algunas tablas extras que no se usarán demasiado.

Comenzaremos la práctica con la explicación de cómo dar de alta la base de datos (squema) en MySQL para posteriormente pasar a los ejercicios.

Video 2- Importación de Base de Datos

Primero que nada, ustedes va a tener disponible una carpeta llamado "Base_Datos", en está se van a encontrar tres archivos: northwindSquema.sql, northWind.sql y northWind-data.sql. Vamos a ver cómo vamos a utilizar cada uno.

Primero, vamos a crear el esquema, y para ello vamos a abrir MySQL Workbench. Me va aparecer la conexión que se hizo previamente. Le damos click y obtendremos una interfaz limpia.

Vamos a darle doble click a nuestro archivo de northwindSquema.sql. Y aquí tanto pueda ser que te de la opción de ejecutarlo cómo que te muestre que no tienes conexión. Puedes ejecutarlo desde ahí o bien, copiar y pegar el contenido en Query1:

```
CREATE DATABASE IF NOT EXISTS northwind;

USE northwind;


DROP TABLE IF EXISTS order_details, orders, products, categories,
                     customers, employees, shippers, suppliers;
```

Este script prepara un entorno limpio en la base de datos northwind. Primero, se asegura de que la base exista, luego la selecciona, y finalmente elimina tablas clave si ya estaban creadas, para poder reconstruirlas desde cero.

Ahora vamos a ver la **estructura de la base de datos.** En esta sección solo daremos de alta las tablas de la base de datos y su estructura, para posteriormente ingresar los datos. Para realizar este proceso tendremos que seleccionar el *squema* northwind y posteriormente ejecutar la *query* de definición del archivo llamado northWind.sql.

```
No escribó aquí el código porque es muy extenso pero este documento lo
tienes a tu disposición.
```

Por último, para esta sección ingresaremos toda la data a nuestras tablas, el proceso será similar a la creación, una vez seleccionado el *squema* northwind ejecutaremos la *query* de manipulación del archivo .sql "northwind-data".

```
No escribó aquí el código porque es muy extenso pero este documento lo
tienes a tu disposición.
```

Con estos tres sencillos pasos ya estamos listos para pasar a la práctica de los temas vistos comenzando por el "Capítulo 2.- Trabajar con una sola tabla".

Video3- Capítulo 2: Trabajar con una sola tabla

Bien, primero que nada van a encontrar un archivo en cada capítulo, en dónde estarán los ejercicios resueltos. Esto para que puedas consultar si te atoras en algo o bien, cuándo hayas terminado.

OJO: Habrá capítulos que no tengan este documento, no es un error, sólo quiere decir que en ese capítulo no habrá ejercicios extra.

Así mismo se encontraran con este manual para mayor información o especificación. Sin más preámbulos, comencemos:

Bien, entonces vamos comenzando:

1. Lista todos los productos activos (no descontinuados) cuyo list_price sea mayor a 25.

```
SELECT product_name, list_price
FROM products
WHERE discontinued = 0 AND list_price > 25;
```

   OJO: La columna de discontinued tiene sólo 0 porque es un campo de tipo bandera booleana, 0 = El producto **NO** está descontinuado (sigue activo); 1 = El producto **sí** está descontinuado.

2. Muestra los productos cuyo nombre contenga la palabra "chocolate", sin importar mayúsculas o minúsculas.

```
SELECT product_name

FROM products

WHERE product_name LIKE '%chocolate%';
```

   OJO: recordemos que la manera en la que escribimos %chocolate% es derivado de lo que aprendimos de expresiones regulares.

3. Lista los productos cuyo precio esté entre 20 y 100, y ordena el resultado por precio descendente.

```
SELECT product_name, list_price

FROM products

WHERE list_price BETWEEN 10 AND 20

ORDER BY list_price DESC;
```

4. Lista todos los productos cuya categoría sea "Beverages", "Condiments" o "Candy".

```
SELECT product_name, category
FROM products
WHERE category IN ('Beverages', 'Condiments', 'Candy');
```

OJO: Quizás te preguntes porqué aquí no usamos REGEX pero ojo, Usamos `LIKE` con `%` porque no sabemos si la palabra "`chocolate`" está al inicio, en medio o al final del nombre. Aquí no buscamos *subcadenas*, sino coincidencias exactas.

5. Muestra los productos con standard_cost menor a 10 o que estén descontinuados.

```
SELECT product_name, standard_cost, discontinued
FROM products
WHERE standard_cost < 10 OR discontinued = 1;
```

OJO: ¿Cuál es la diferencia entre este ejercicio y el ejercicio 1? Aquí si estamos añadiendo la columna "`discontinued`" en `SELECT`. Pero ¿por qué? El filtro del ejercicio 1 dice: *solo productos activos (discontinued = 0) y con precio mayor a 25*. Entonces, tu ya sabes que todos los productos que salgan cumplen esa condición, entonces no necesitas ver la columna "`discontinued`**"**. En cambio, aquí el filtro tiene **dos caminos**: productos baratos (`standard_cost < 10`), **o** productos descontinuados (`discontinued = 1`). Si no incluyes la columna `discontinued` en el `SELECT`, no sabrías **por qué** salió cada producto.

6. Encuentra productos que no tengan descripción registrada.

```
SELECT product_name
FROM products
WHERE description IS NULL;
```

7. Obtén los 5 productos más caros (list_price) que estén activos.

```
SELECT product_name, list_price
FROM products
WHERE discontinued = 0
ORDER BY list_price DESC
LIMIT 5;
```

8. Muestra los productos cuyo nombre termina en “Tea”.

```
SELECT product_name

FROM products

WHERE product_name LIKE '%Tea';
```

9. Lista los productos con categoría distinta de "Oil", "Candy" y "Cereal" y con list_price mayor a 15.

```
SELECT product_name, category, list_price

FROM products

WHERE category NOT IN ('Oil', 'Candy', 'Cereal') AND
list_price > 15;
```

10. Muestra los productos activos cuyo product_name solo contenga letras y espacios (sin números). Usa una expresión regular.

```
SELECT product_name

FROM products

WHERE discontinued = 0 AND product_name REGEXP '^[A-Za-z ]+$';
```

OJO: La expresión regular `'^[A-Za-z ]+$'` vamos a desmenuzarla:

- `^` → inicio de la cadena.
- `[A-Za-z ]+` → uno o más caracteres que sean:
  - letras mayúsculas (`A-Z`)
  - letras minúsculas (`a-z`)
  - espacios ( ).
- `$` → final de la cadena

Video 4- Capítulo 3: Trabajar con Varias Tablas

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap3Ejercicios.sql".

1. Muestra el nombre del producto y el nombre del proveedor correspondiente para todos los productos activos (no descontinuados).

```
SELECT p.product_name, s.company AS supplier_company
FROM products p
JOIN suppliers s ON p.supplier_ids = s.id
WHERE p.discontinued = 0;
```

   OJO: Tenemos que especificar con que tabla estamos trabajando porque ya no estamos manejando sólo una cómo en el capítulo pasado y las columnas se pueden confundir. Así mismo, pusimos alías a las tablas de products y suppliers para poder acortar la notación. En lugar de poner "`products`.`product_name`" solo ponemos "`p.product_name`"

   OJO: La condición de unión es que el campo `supplier_ids` en `products` coincida con `id` en `suppliers`.

2. Muestra el nombre del producto, su categoría y el proveedor solo si su precio de lista es mayor a 20 y pertenece a las categorías 'Condiments' o 'Candy'.

```
SELECT p.product_name, p.category, p.list_price, s.company
AS supplier_company
FROM products p
JOIN suppliers s ON p.supplier_ids = s.id
WHERE p.category IN ('Condiments', 'Candy')
 AND p.list_price > 20;
```

3. Selecciona el nombre de la empresa del cliente y el id de las órdenes y filtra únicamente aquellas empresas cuyo nombre empiece con la letra 'C'

```
SELECT c.company AS customer_company, o.id AS order_id
FROM customers c
JOIN orders o ON c.id = o.customer_id
WHERE c.company LIKE 'C%';
```

OJO: Tú empiezas desde la tabla `customers`. Después dices: "tráeme también datos de la tabla `orders`, siempre que coincidan". La condición de unión es: `c.id = o.customer_id`

- `c.id` = clave primaria de `customers`.
- `o.customer_id` = clave foránea en `orders` que apunta a `customers`.

4. Muestra los pares de empleados que trabajan en la misma ciudad. Evita mostrar combinaciones repetidas o un empleado comparado consigo mismo. Ordena el resultado por el nombre del primer empleado.

```
SELECT
  e1.first_name AS empleado_1,
  e1.last_name AS apellido_1,
  e2.first_name AS empleado_2,
  e2.last_name AS apellido_2,
  e1.city
FROM employees e1
JOIN employees e2 ON e1.city = e2.city AND e1.id < e2.id
ORDER BY e1.first_name;
```

OJO: dos alias distintos de la misma tabla `employees`. Piensa en ello como si tuvieras "dos copias" de la tabla y las cruzaras entre sí.

`e1.city = e2.city`: empareja empleados que viven en la misma ciudad.

`e1.id < e2.id`: asegura que no se empareje el mismo empleado consigo mismo ni se repita el par en orden inverso.

- Sin esto, tendrías (A, B) y (B, A).
- Con esto, solo tienes (A, B).

`ORDER BY e1.first_name` → ordena el resultado según el primer nombre del primer empleado en cada par.

5. Lista todos los productos y su proveedor si tienen uno. Si no tienen proveedor asignado, indica 'SIN PROVEEDOR'.

```
 SELECT p.product_name, COALESCE(s.company, 'SIN PROVEEDOR')
AS supplier_company
```

```
FROM products p
LEFT JOIN suppliers s ON p.supplier_ids = s.id;
```

OJO:

`COALESCE` dice: Si `s.company` existe → úsalo. Si es `NULL` → pon la frase `'SIN PROVEEDOR'`. Así evitas ver `NULL` en tu resultado.

Toma todos los productos de la tabla `products`.Intenta buscar su proveedor en la tabla `suppliers` comparando `p.supplier_ids = s.id`.

Como es **LEFT JOIN**, si un producto no tiene proveedor, de todas formas aparece en la lista (pero con `NULL` en los datos del proveedor, a excepción de si usaste Coalesce para decir lo contrario).

6. Muestra todos los proveedores, incluyendo aquellos que no tienen productos asociados.

```
SELECT s.company AS supplier_company, p.product_name
FROM suppliers s
LEFT JOIN products p ON p.supplier_ids = s.id;
```

`LEFT JOIN` entonces significa:

- Toma **todos los registros de la tabla de la izquierda** (`suppliers`).
- Une los productos (`products`) que coincidan en la condición `p.supplier_ids = s.id`.
- Si un proveedor **no tiene productos**, igual aparece, pero `p.product_name` será `NULL`.

7. Genera combinaciones de productos con proveedores para explorar posibles asignaciones. Limita el resultado a 20.

```
SELECT p.product_name, s.company AS supplier_company
FROM products p
CROSS JOIN suppliers s
LIMIT 20;
```

- **OJO**: Un `CROSS JOIN` produce el producto cartesiano de ambas tablas. Es decir: combina cada fila de `products` con cada fila de `suppliers`.

No hay condición de unión (como `ON …`), así que se generan todas las combinaciones posibles.

Como el resultado puede ser enorme (si tienes 50 productos y 10 proveedores → 500 combinaciones), el `LIMIT 20` corta el resultado para mostrar solo 20 filas.

8. Muestra clientes localizados en los estados NY o MN. Evita duplicados.

```
SELECT company, state_province
 FROM customers
 WHERE state_province = 'NY'
 UNION
 SELECT company, state_province
 FROM customers
 WHERE state_province = 'MN';
```

También se puede hacer de esta forma pero NO elimina duplicados (en este caso no importa pero habrá otros escenarios en dónde si)

```
SELECT company, state_province
FROM customers
WHERE state_province IN ('NY','MN');
```

9. Muestra el nombre del producto, su precio y la cantidad pedida en cada línea de pedido.

```
 SELECT p.product_name, p.list_price, od.quantity
 FROM order_details od
 JOIN products p ON od.product_id = p.id;
```

10. Muestra el nombre del cliente, el ID del pedido y el empleado que gestionó el pedido.

```
 SELECT c.company AS cliente, o.id AS order_id, e.first_name
AS empleado
 FROM orders o
 JOIN customers c ON o.customer_id = c.id
 JOIN employees e ON o.employee_id = e.id;
```

Video 5- Capítulo 4: Trabajar con Datos

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap4Ejercicios.sql".

1. Ejercicio: Explica brevemente qué hace cada uno de los siguientes atributos al crear una columna:
   - `DEFAULT:` Asigna un valor por defecto a la columna cuando insertas una fila y no especificas un valor para esa columna. Por ejemplo, si no indicas precio al insertar, se pondrá automáticamente 0.00.
   - `AUTO_INCREMENT:` Genera automáticamente valores consecutivos (1, 2, 3, …) para esa columna cada vez que insertas una nueva fila. Solo puede aplicarse a una columna de tipo entero. Normalmente se usa en la clave primaria (PRIMARY KEY)**.** Cada vez que insertes una fila, MySQL asigna un `id` nuevo.
   - `NOT NULL:` Obliga a que la columna no acepte valores nulos (`NULL`). Por ejemplo: No puedes insertar un producto sin nombre.
   - `UNIQUE:` Garantiza que los valores de esa columna no se repitan entre filas.No pueden existir dos clientes con el mismo email.

2. Actualiza el nombre del contacto (first_name) del cliente con id = 6 a 'Luis'.

```
SELECT id, first_name, last_name
FROM customers
WHERE id = 6;


UPDATE customers
SET first_name = 'Luis'
WHERE id = 6;
```

3. Simula que el empleado Jan, quien actualmente tiene el puesto de 'Sales Representative', ha sido ascendido a 'Sales Manager'.

   Actualiza su título en la tabla employees.

```
UPDATE employees
SET job_title = 'Sales Manager'
```

```
WHERE first_name = 'Jan' AND job_title = 'Sales
Representative';
```

4. Los proveedores no tienen registrado el estado (state_province) en el que se encuentran. Ingresa los siguientes datos manualmente en la tabla suppliers:
   - Supplier A | NY
   - Supplier B | CA
   - Supplier C | NY
   - Supplier D | TX
   - Supplier E | TX
   - Supplier F | WA
   - Supplier G | CA
   - Supplier H | WA
   - Supplier I | NY
   - Supplier J | TX

```
UPDATE suppliers

SET state_province = 'NY' WHERE company = 'Supplier A';

UPDATE suppliers SET state_province = 'CA' WHERE company =
'Supplier B';

UPDATE suppliers SET state_province = 'NY' WHERE company =
'Supplier C';

UPDATE suppliers SET state_province = 'TX' WHERE company =
'Supplier D';

UPDATE suppliers SET state_province = 'TX' WHERE company =
'Supplier E';

UPDATE suppliers SET state_province = 'WA' WHERE company =
'Supplier F';

UPDATE suppliers SET state_province = 'CA' WHERE company =
'Supplier G';

UPDATE suppliers SET state_province = 'WA' WHERE company =
'Supplier H';

UPDATE suppliers SET state_province = 'NY' WHERE company =
'Supplier I';
```

```
UPDATE suppliers SET state_province = 'TX' WHERE company =
'Supplier J';
```

5. Crea una nueva tabla products_backup con todos los productos.

```
CREATE TABLE products_backup AS

SELECT *

FROM products
```

"*" es un comodín que indica "todo".

OJO: La nueva tabla tendrá **l**os mismos datos**,** pero no hereda claves primarias, índices, ni restricciones (solo copia la estructura básica y los valores).

6. Ahora que ya se completaron los estados, la empresa desea ofrecer un descuento en costos de producción a los productos cuyos proveedores estén ubicados en el estado de New York (NY). Reduce en un 10% el standard_cost de todos los productos cuyo proveedor (suppliers) esté ubicado en el estado 'NY'.

```
SELECT p.product_name, p.standard_cost, s.company,
s.state_province

FROM products p

JOIN suppliers s ON p.supplier_ids = s.id

WHERE s.state_province = 'NY';


UPDATE products p

JOIN suppliers s ON p.supplier_ids = s.id

SET p.standard_cost = p.standard_cost * 0.9

WHERE s.state_province = 'NY';
```

OJO: Si se desea hacer un UPDATE sin una cláusula WHERE que involucre una clave primaria el software MySQL no lo permitirá con el objetivo de prevenir la eliminación de bases de datos, sin embargo, esta configuración "segura" puede ser eliminada mediante el comando:

```
SET SQL_SAFE_UPDATES = 0;
```

7. Aumenta el list_price en 5% para todos los productos de la categoría 'Beverages' que tengan minimum_reorder_quantity mayor o igual a 25.

```
UPDATE products
SET list_price = list_price * 1.05
WHERE category = 'Beverages'
  AND minimum_reorder_quantity >= 25;
```

8. Actualiza el campo taxes de la tabla orders, asignándole un 16% del shipping_fee solo a los pedidos cuyo shipping_fee sea mayor al promedio general de todos los pedidos.

   NOTA: Para resolver esta consulta es necesario utilizar una tabla derivada.

```
UPDATE orders
JOIN (
    SELECT AVG(shipping_fee) AS avg_shipping
    FROM orders
) AS avg_table
SET taxes = shipping_fee * 0.16
WHERE shipping_fee > avg_table.avg_shipping;
```

   Si quieres ver el promedio poner:

```
SELECT AVG(shipping_fee) AS avg_shipping
FROM orders;
```

   OJO: En este caso también se podría hacer con un WHERE pero devolvería un valor escalar y estamos practicando las tablas derivadas:

```
…SELECT o.order_id, o.shipping_fee
FROM orders o
WHERE o.shipping_fee > (
    SELECT AVG(shipping_fee)
    FROM orders
)…
```

9. Elimina los productos que han sido descontinuados (discontinued = 1) y que no aparecen en ninguna línea de pedido (order_details).

```
DELETE FROM products
WHERE discontinued = 1
  AND id NOT IN (
    SELECT DISTINCT product_id
    FROM order_details
  );
```

OJO: `SELECT DISTINCT product_id`: Tráeme todos los valores de la columna `product_id`, pero sin repetirlos**.**

A continuación, se presenta un ejercicio un poco más complicado ya que involucra temas que veremos en el siguiente capítulo, con esto se espera que se introduzca el funcionamiento de GROUP BY y que al mismo tiempo se aborde el cómo realizar una actualización con una operación de agregación y un JOIN.

10. Un proveedor desea ajustar sus precios en función del volumen de ventas. Actualiza el list_price de cada producto incrementándolo en un 5% si ese producto ha vendido más de 100 unidades en total (sumando todas las órdenes en order_details).

    Nota: Usa UPDATE con JOIN y subconsulta agregada.

    ---Si quieres ver qué se afectará antes ponemos:

```
SELECT p.id, p.product_name, p.list_price, ventas.total_vendido
FROM products p
JOIN (
    SELECT product_id, SUM(quantity) AS total_vendido
    FROM order_details
    GROUP BY product_id
    HAVING total_vendido > 100
) AS ventas
ON p.id = ventas.product_id;
```

```
UPDATE products p
JOIN (
    SELECT product_id, SUM(quantity) AS total_vendido
    FROM order_details
    GROUP BY product_id
    HAVING total_vendido > 100
) AS ventas ON p.id = ventas.product_id
SET p.list_price = p.list_price * 1.05;
```

OPCIÓN 2 DE QUITAR EL MODO SEGURO-

Desactivar Safe Update Mode en Workbench (MAC)

1. Ve a **Preferences → SQL Editor**.
2. Quita la palomita en **"Safe Updates (rejects UPDATEs and DELETEs with no key in WHERE clause)"**.
3. Cierra sesión y vuelve a conectar.
   Esto te permite ejecutar tu `UPDATE` tal cual.

Desactivar *Safe Update Mode* en MySQL Workbench (Windows)

1. **Abre MySQL Workbench**.
2. Arriba a la izquierda, ve al menú:
   Edit → Preferences…
3. En la ventana de preferencias, en el panel izquierdo selecciona:
   SQL Editor.
4. En la sección **"Other"** (o "SQL Execution" en algunas versiones), busca la opción:
   Safe Updates (rejects UPDATEs and DELETEs with no key in WHERE clause).
5. **Desmarca la casilla**.
6. Haz clic en **OK** para guardar los cambios.
7. **Cierra la conexión** actual (o incluso el Workbench) y **vuelve a conectar** a tu servidor.

Video 6- Capítulo 5: Resumir Datos

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap5Ejercicios.sql".

1. Muestra el número total de órdenes realizadas por cada cliente. Incluye el nombre de la empresa (company) y ordena el resultado de mayor a menor.

```
SELECT c.company, COUNT(o.id) AS total_ordenes
FROM customers c
JOIN orders o ON c.id = o.customer_id
GROUP BY c.company
ORDER BY total_ordenes DESC;
```

   OJO: `GROUP BY` , agrupa filas que comparten el mismo valor en X columna.

   En este caso, agrupa todas las filas por cliente (company) y calcula la función de agregación (`COUNT`) para cada grupo.

2. Muestra cuántos pedidos se han enviado desde cada ciudad (ship_city). Ordena de mayor a menor.

```
SELECT ship_city, COUNT(*) AS total_pedidos
FROM orders
GROUP BY ship_city
ORDER BY total_pedidos DESC;
```

   OJO: `COUNT(*)` cuenta todas las filas, sin importar si hay `NULL` o qué columnas seleccionas. `COUNT(o.id)` cuenta únicamente las filas donde `o.id` no sea NULL y eso funciona porque `id` es normalmente la clave primaria de `orders`, (nunca debería ser `NULL`). Aunque bueno, en este caso, `COUNT(*)` y `COUNT(o.id)` darán el mismo resultado.

3. Muestra el nombre de cada producto y el promedio de unidades (quantity) vendidas por orden. Solo incluye productos vendidos en al menos 5 órdenes.
   *Tomar Ejemplo Manual Con El Id-43

```
SELECT p.product_name, AVG(od.quantity) AS promedio_vendido
FROM products p
```

```
JOIN order_details od ON p.id = od.product_id
GROUP BY p.product_name
HAVING COUNT(od.id) >= 5;
```

--Si quieres que también te salga el id:

```
SELECT p.id,
       p.product_name,
       AVG(od.quantity) AS promedio_vendido
FROM products p
JOIN order_details od ON p.id = od.product_id
GROUP BY p.id, p.product_name
HAVING COUNT(od.id) >= 5;
```

4. Muestra cuántos pedidos ha gestionado cada empleado (employee_id). Muestra solo aquellos con más de 3 pedidos.

```
SELECT employee_id, COUNT(*) AS total_pedidos
FROM orders
GROUP BY employee_id
HAVING total_pedidos > 3;
```

5. Muestra cada producto con su cantidad total vendida (SUM(quantity)), su precio unitario promedio (en otros casos el precio varia entre órdenes) y el ingreso total generado (SUM(quantity * unit_price)).

```
SELECT p.product_name,
       SUM(od.quantity) AS total_vendido,
       AVG(od.unit_price) AS precio_promedio,
       SUM(od.quantity * od.unit_price) AS ingreso_total
FROM products p
JOIN order_details od ON p.id = od.product_id
GROUP BY p.product_name;
```

6. Agrupa por estado (ship_state_province) y muestra el total de shipping_fee e impuestos (taxes) recaudados por estado.

```
SELECT ship_state_province,
       SUM(shipping_fee) AS total_tarifa_envios,
       SUM(taxes) AS total_impuestos
FROM orders
GROUP BY ship_state_province;
```

--- Algunos estarán en 0 porque nosotros establecimos los taxes de la tabla orders, asignándole un 16% del shipping_fee solo a los pedidos cuyo shipping_fee sea mayor al promedio general de todos los pedidos (Ejercicio del Cap 4)

7. Muestra el nombre de cada producto y el descuento promedio aplicado. Incluye solo productos con al menos 3 ventas.

NOTA: Ninguna de nuestras ordenes fue realizada con descuento por lo que esperamos obtener una columna de solo 0.

```
SELECT p.product_name, AVG(od.discount) AS descuento_promedio
FROM products p
JOIN order_details od ON p.id = od.product_id
GROUP BY p.product_name
HAVING COUNT(od.id) >= 3;
```

8. Muestra el número total de productos registrados por estado (state_province) de su proveedor. Incluye un total general usando WITH ROLLUP.

```
SELECT s.state_province, COUNT(p.id) AS total_productos
FROM products p
JOIN suppliers s ON p.supplier_ids = s.id
GROUP BY s.state_province WITH ROLLUP;
```

OJO: `WITH ROLLUP` dice, *además de los totales por grupo, agrega una fila extra con el* ***gran total***. Esa fila extra tendrá `NULL` en la columna agrupada (`state_province`) y el total de todos los productos en `total_productos`.

9. Muestra el total de unidades vendidas (SUM(quantity)) por producto y añade un total general al final con WITH ROLLUP.

```
SELECT p.product_name, SUM(od.quantity) AS total_vendido
FROM products p
JOIN order_details od ON p.id = od.product_id
GROUP BY p.product_name WITH ROLLUP;
```

10. Muestra el nombre de cada producto y el total de unidades vendidas (SUM(quantity)). Solo incluye productos con más de 100 unidades vendidas.

```
SELECT p.product_name, SUM(od.quantity) AS total_vendido
FROM products p
JOIN order_details od ON p.id = od.product_id
GROUP BY p.product_name
HAVING total_vendido > 100;
```

Video 7- Capítulo 6: Queries Complejos

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap6Ejercicios.sql".

1. Lista los productos cuyo list_price es mayor que el promedio de su misma categoría. Utiliza una subconsulta.

```
SELECT p1.product_name, p1.category, p1.list_price
FROM products p1
WHERE p1.list_price >
      (SELECT AVG(p2.list_price)
       FROM products p2
       WHERE p2.category = p1.category);
```

- **OJO**: Una subconsulta es análoga a un cálculo dentro de una fórmula en Excel**:** algo que se resuelve primero y luego se compara/usa en la expresión principal.

  OJO:

  - `p2` es otro alias de la misma tabla `products`.
  - `AVG(p2.list_price)` obtiene el promedio de precios.
  - La condición `WHERE p2.category = p1.category` asegura que el promedio se calcule únicamente con los productos de **esa categoría específica**.

2. Muestra las empresas (company) de clientes que sí han hecho al menos una orden. Utiliza IN.

```
SELECT c.company
FROM customers c
WHERE c.id IN (SELECT o.customer_id FROM orders o);
```

- **OJO:** `WHERE c.id IN (…)`**:** pide que solo se incluyan aquellos clientes cuyo `id` aparezca en la lista que devuelve la subconsulta.

  `SELECT o.customer_id FROM orders o`**:** devuelve todos los `customer_id` que existen en la tabla `orders`, o sea, los clientes que han hecho pedidos.

Join vs subquery

3. Total de órdenes por cliente (muestra `company` y `total_ordenes`).

Utilizando JOIN Directo:

```
SELECT c.company, COUNT(o.id) AS total_ordenes
FROM customers c
JOIN orders o ON o.customer_id = c.id
GROUP BY c.company;
```

Utilizando subconsulta:

```
SELECT c.company, t.total_ordenes
FROM customers c
JOIN (
    SELECT customer_id, COUNT(*) AS total_ordenes
    FROM orders
    GROUP BY customer_id
) AS t ON t.customer_id = c.id;
```

JOIN directo	Subconsulta
Hace todo en un solo paso (JOIN + GROUP BY).	Divide en dos pasos: primero calcula la tabla de totales, luego la une.
Menos código, más compacto.	Más modular y explícito (puedes reutilizar o extender la subconsulta).
Depende de agrupar por un campo de `customers` (`company`).	Agrupa solo `orders` y luego se une.

4. Muestra, por cada proveedor, el producto cuyo `list_price` es el más alto. Usa el operador `ALL` para compararlo con los precios de los demás productos del mismo proveedor.

```
SELECT p.product_name, p.list_price, p.supplier_ids
```

```
FROM products p
WHERE p.list_price > ALL (
    SELECT p2.list_price
    FROM products p2
    WHERE p2.supplier_ids = p.supplier_ids
      AND p2.id <> p.id
);
```

OJO: "`AND p2.id <> p.id`": Esta diciendo que se Excluye el producto actual de la lista de comparación". El operador <> es "distinto de". Si no pusiéramos esta condición, el producto se estaría comparando contra sí mismo (y siempre sería igual a su propio precio).

5. Muestra las órdenes cuyo shipping_fee sea mayor que cualquiera de las shipping_fee de las órdenes enviadas al estado 'NY' (en la práctica: `> ANY` ≈ mayor que el mínimo de esa lista, es decir, mayor que el mínimo del estado de 'NY'). Utiliza ANY.

```
SELECT id, ship_city, ship_state_province, shipping_fee
FROM orders
WHERE shipping_fee IS NOT NULL
  AND shipping_fee > ANY (
    SELECT o2.shipping_fee
    FROM orders o2
    WHERE o2.ship_state_province = 'NY'
      AND o2.shipping_fee IS NOT NULL
);
```

- **OJO**: En este caso, NY tiene un `shipping_fee` mínimo de 0, entonces aquellos `shipping_fee` iguales a 0 se descartarán porque estamos manejando "`>0`".

OJO: Cuando usas comparaciones con `ANY` o `ALL` en SQL Comparar con un número funciona bien (`15 > 10`) pero comparar con NULL siempre devuelve `UNKNOWN` (ni verdadero ni falso).

6. Muestra órdenes cuyo shipping_fee sea mayor que el promedio de envío de ese mismo cliente.

```
SELECT o.*
FROM orders o
WHERE o.shipping_fee >
      (SELECT AVG(o2.shipping_fee)
       FROM orders o2
       WHERE o2.customer_id = o.customer_id);
```

- **OJO:** Tomar ejemplo de explicación del customer_id (Por ejemplo, del customer_id = 4, hay 5 órdenes. De esas 5 órdenes se saca el promedio (que es 3.6) y sólo se descarta una orden con shipping_fee=0, que es menor a 3.6).

7. Lista productos que nunca han sido vendidos (no aparecen en order_details).

```
SELECT p.id, p.product_name
FROM products p
WHERE NOT EXISTS (
    SELECT 1
    FROM order_details od
    WHERE od.product_id = p.id
);
```

- **OJO**: Dentro de `EXISTS` / `NOT EXISTS`, lo que pongas después de `SELECT` no se usa en el resultado. Solo se comprueba si la subconsulta devuelve alguna fila o ninguna. Por convención, se suele escribir `SELECT 1` porque es corto y deja claro que no importa el valor.

8. Para cada cliente, muestra company, número de órdenes y suma de shipping_fee. Utiliza consulta dentro de la cláusula SELECT.

```
SELECT c.company,
```

```
       (SELECT COUNT(*) FROM orders o WHERE o.customer_id =
c.id) AS total_ordenes,

       (SELECT SUM(o.shipping_fee) FROM orders o WHERE
o.customer_id = c.id) AS total_costo_envio

FROM customers c;
```

- **OJO:** Count(*) cuenta todas las filas que cumplen la condición. SUM(o.shipping_fee) suma el contenido de las filas.

9. Calcula el total de unidades vendidas por producto en una tabla derivada y muestra solo los que están por encima del promedio global. Utiliza subconsulta en la cláusula FROM.

```
SELECT p.product_name, v.total_unidades

FROM (

    SELECT product_id, SUM(quantity) AS total_unidades

    FROM order_details

    GROUP BY product_id

) AS v

JOIN products p ON p.id = v.product_id

WHERE v.total_unidades > (

    SELECT AVG(t.total_unidades)

    FROM (

        SELECT SUM(quantity) AS total_unidades

        FROM order_details

        GROUP BY product_id

    ) AS t

);
```

Si quieres checar cuál es el promedio:

```
SELECT AVG(t.total_unidades) AS promedio_general

FROM (
```

```
    SELECT product_id, SUM(quantity) AS total_unidades
    FROM order_details
    GROUP BY product_id
) AS t;
```

OJO: Tomar cómo ejemplo el producto 34 (N T Beer) en order_details en dónde se ve que el total de unidades es 487.

10. Muestra proveedores que tienen al menos un producto que se haya vendido una cantidad mayor o igual a 50 unidades en una sola orden.

```
SELECT DISTINCT s.company, s.state_province
FROM suppliers s
WHERE EXISTS (
    SELECT 1
    FROM products p
    JOIN order_details od ON od.product_id = p.id
    WHERE p.supplier_ids = s.id
      AND od.quantity >= 50);
```

- `DISTINCT` asegura que, si un proveedor cumple varias veces la condición, salga solo una vez en el resultado.

Video 8- Capítulo 7: Funciones Esenciales de SQL

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap7Ejercicios.sql".

1. Calcula, para cada producto, el margen de ganancia (list_price − standard_cost) y el porcentaje de margen de ganancia redondeado a 2 decimales. Muestra solo productos con list_price > 0. Utiliza funciones numéricas.

```
SELECT
  p.product_name,
  p.category,
  ROUND(p.list_price - p.standard_cost, 2) AS margen,
  ROUND((p.list_price - p.standard_cost) / p.list_price *
100, 2) AS porcentaje_margen
FROM products p
WHERE p.list_price > 0
```

2. Muestra product_name, list_price, el precio con IVA 16% redondeado a 2 decimales, además de CEIL y FLOOR de ese precio.

```
SELECT
  p.product_name,
  p.list_price,
  ROUND(p.list_price * 1.16, 2) AS precio_iva,
  CEIL(p.list_price * 1.16)     AS precio_techo,
  FLOOR(p.list_price * 1.16)    AS precio_piso
FROM products p;
```

   - **OJO**: `CEIL` (**ceiling**) redondea hacia arriba al entero más cercano. `FLOOR` (**floor**) redondea **hacia abajo** al entero más cercano.

3. Genera un email sugerido nombre.apellido@empresa.local en minúsculas, sin espacios. Utiliza funciones de texto.

```
SELECT
  e.first_name,
```

```
  e.last_name,
  CONCAT(
    LOWER(REPLACE(e.first_name, ' ', '')),
    '.',
    LOWER(REPLACE(e.last_name,  ' ', '')),
    '@empresa.local'
  ) AS email_sugerido
FROM employees e;
```

- **OJO:**
  - `REPLACE(e.first_name, ' ', '')`: Quita espacios en caso de que el nombre o apellido tengan varios.
  - `LOWER(...)`:Convierte todo a minúscula
  - `CONCAT(...)`: Une varias partes en una sola cadena:

4. Realiza la inserción de correos para los proveedores (suppliers). Posteriormente como ejercicio extrae el dominio de email_address y cuenta cuántos proveedores tiene cada dominio (ordena de mayor a menor). Utiliza funciones de texto.

```
-- SET SQL_SAFE_UPDATES = 0;
UPDATE suppliers s
SET   s.email_address   =   CONCAT('contact',   s.id,   '@',
LOWER(REPLACE(s.company, ' ', '')), '.com')
WHERE (s.email_address IS NULL OR s.email_address = '');


SELECT
  LOWER(SUBSTRING_INDEX(TRIM(s.email_address), '@', -1)) AS
dominio,
  COUNT(*) AS total_proveedores
FROM suppliers s
WHERE s.email_address IS NOT NULL
  AND s.email_address <> ''
```

```
  AND INSTR(s.email_address, '@') > 0
GROUP BY dominio
ORDER BY total_proveedores DESC;
```

- **OJO**:
  - `TRIM(s.email_address)` : Quita espacios al inicio/fin.
  - `SUBSTRING_INDEX(…, '@', -1)`: Parte el texto por `'@'` y devuelve lo que está después del último `@` (`-1` = "después del último separador")
  - `LOWER(…) AS dominio`: Normaliza a minúsculas para que `Gmail.com` y `gmail.com` cuenten como el mismo dominio.
  - `COUNT(*) AS total_proveedores`: cuenta cuántas filas (proveedores) caen en cada dominio.
  - `WHERE …` filtros de higiene
    - `IS NOT NULL` y `<> ''` → evita nulos y vacíos.
    - `INSTR(email_address, '@') > 0` → exige que exista un `@` (básico para considerarlo email).
  - `GROUP BY dominio`: Agrupa por el dominio extraído para poder contar.
  - `ORDER BY total_proveedores DESC`: Muestra primero los dominios más frecuentes.

5. Calcula DATEDIFF(shipped_date, order_date) como dias_envio. Muestra solo órdenes con shipped_date no nulo. Utiliza funciones de fecha.

```
SELECT
  o.id,
  o.order_date,
  o.shipped_date,
  DATEDIFF(o.shipped_date, o.order_date) AS dias_envio
FROM orders o
WHERE o.shipped_date IS NOT NULL
ORDER BY dias_envio DESC;
```

- **OJO**: En `DATEDIFF` El orden importa**,** solo cuenta días completos (no incluye horas ni segundos) y cuenta días naturales.

6. Muestra año, mes y número de pedidos realizados en ese mes. Utiliza funciones de fecha.

```
SELECT
  YEAR(o.order_date)  AS anio,
  MONTH(o.order_date) AS mes,
  COUNT(*)            AS total_pedidos
FROM orders o
GROUP BY YEAR(o.order_date), MONTH(o.order_date)
ORDER BY anio, mes;
```

- **OJO**:
  - `GROUP BY YEAR(o.order_date), MONTH(o.order_date)`: Agrupa todos los pedidos por combinación de año + mes**.**
  - `ORDER BY anio, mes`: Ordena el resultado cronológicamente, primero 2024, luego 2025, etc., y dentro de cada año, de enero a diciembre.

7. Muestra shipping_fee, taxes y el total_extra usando COALESCE para tratar nulos.

```
SELECT
  o.id,
  COALESCE(o.shipping_fee, 0) AS shipping_fee_efectivo,
  COALESCE(o.taxes, 0)        AS taxes_efectivo,
  COALESCE(o.shipping_fee, 0) + COALESCE(o.taxes, 0) AS
total_extra
FROM orders o
ORDER BY total_extra DESC;
```

- **OJO:**
  - `COALESCE(expr, valor)` devuelve el primer valor que no sea NULL.
    - Si `shipping_fee` tiene un número, devuelve ese número. Si es `NULL`, devuelve `0` entonces siempre tendrás un número para el cargo de envío.
  - `COALESCE(o.taxes, 0) AS taxes_efectivo`: Lo mismo, pero aplicado a la columna de impuestos.
  - `COALESCE(o.shipping_fee, 0) + COALESCE(o.taxes, 0) AS total_extra`: Suma ambos valores (ya sin `NULL`), es decir: total adicional que el cliente paga aparte del precio del producto.

8. Muestra cada order_details con un indicador “Con descuento / Sin descuento”. Utiliza la función SI (IF).

```
SELECT
  od.id,
  od.order_id,
  od.product_id,
  od.quantity,
  od.unit_price,
  od.discount,
  IF(od.discount > 0, 'Con descuento', 'Sin descuento') AS aplica_descuento
FROM order_details od;
```

9. Clasifica shipping_fee en Bajo (<20), Medio (20–99.99) y Alto (≥100). Muestra también cuántos pedidos hay en cada clase.

```
SELECT
  CASE
    WHEN o.shipping_fee >= 100 THEN 'Alto'
    WHEN o.shipping_fee >= 20  THEN 'Medio'
    ELSE 'Bajo'
  END AS rango_envio,
  COUNT(*) AS total_pedidos
FROM orders o
GROUP BY rango_envio
ORDER BY total_pedidos DESC;
```

- **OJO:**
  - En SQL, `CASE` es una **estructura condicional** que funciona parecido a un `IF…ELSE` en otros lenguajes de programación.
  - `WHEN … THEN …` → cada condición que SQL evalúa en orden.

- `ELSE` → valor que se devuelve si ninguna condición se cumple (opcional, si lo omites y nada coincide → devuelve `NULL`).
- `END` → cierra el bloque.

El valor que devuelve `CASE` se puede usar en un `SELECT`, en un `WHERE`, en un `ORDER BY`, etc.

10. Para cada cliente, muestra company en mayúsculas, fecha de última orden y total de envío acumulado, manejando nulos con COALESCE.

```
SELECT
  UPPER(c.company) AS company_mayus,
  (SELECT MAX(o.order_date) FROM orders o WHERE o.customer_id = c.id) AS ultima_orden,
  (SELECT ROUND(SUM(COALESCE(o.shipping_fee, 0)), 2) FROM orders o WHERE o.customer_id = c.id) AS total_envio
FROM customers c
ORDER BY total_envio DESC;
```

- OJO:

Subconsulta 1 → **`ultima_orden`**

- Busca las órdenes del cliente (`o.customer_id = c.id`).
- `MAX(o.order_date)` devuelve la **fecha más reciente** de pedido.
- Resultado: para cada cliente, obtienes su última orden.

Subconsulta 2 → `total_envio`

- Suma (`SUM`) todos los `shipping_fee` (costos de envío) de las órdenes de ese cliente.
- `COALESCE(o.shipping_fee, 0)` si algún envío es `NULL`, lo convierte a `0` para que no rompa la suma.
- `ROUND(..., 2)` redondea el total a 2 decimales.
- Resultado: total de gastos en envío por cliente.

Video 9- Capítulo 8: Vistas

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap8Ejercicios.sql".

1. Crea una vista con el total de órdenes, total de envío, total de impuestos y fecha de última orden por cliente. Luego consúltala ordenando por total_ordenes.

```
-- Crear / reemplazar
CREATE OR REPLACE VIEW vw_cliente_resumen_pedidos AS
SELECT
  c.id              AS customer_id,
  c.company,
  COUNT(o.id)                              AS total_ordenes,
  ROUND(SUM(COALESCE(o.shipping_fee, 0)), 2) AS total_envio,
  ROUND(SUM(COALESCE(o.taxes,  0)),  2)                AS
total_impuestos,
  MAX(o.order_date)                          AS ultima_orden
FROM customers c
JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.company;


-- Consultar la vista cómo si fuera una tabla normal
SELECT * FROM vw_cliente_resumen_pedidos
ORDER BY total_ordenes DESC;
```

OJO: Recordemos que una vista es como una consulta guardada: no almacena datos nuevos, sino que cada vez que la consultas, ejecuta el `SELECT`. Si ya existía, la reemplaza.

`GROUP BY c.id, c.company`: Agrupa la información por cliente → 1 fila por cliente.

2. Crea una vista con unidades totales vendidas, ingreso total y número de órdenes distintas por producto. Luego consulta los productos con más de 100 unidades.

```
-- Crear / reemplazar
CREATE OR REPLACE VIEW vw_producto_ventas AS
SELECT
  p.id AS product_id,
  p.product_name,
  SUM(od.quantity)                                AS
total_unidades,
  ROUND(SUM(od.quantity * od.unit_price), 2)      AS
ingreso_total,
  COUNT(DISTINCT od.order_id)                     AS
ordenes_distintas
FROM products p
JOIN order_details od ON od.product_id = p.id
GROUP BY p.id, p.product_name;


-- Consultar filtro > 100 unidades
SELECT product_name, total_unidades, ingreso_total
FROM vw_producto_ventas
WHERE total_unidades > 100
ORDER BY ingreso_total DESC;
```

3. Crea una vista con employee_id, nombre, año, mes y total de pedidos del mes. Luego consulta el contenido ordenado por año/mes.

```
-- Crear / reemplazar
CREATE OR REPLACE VIEW vw_empleado_pedidos_mensuales AS
SELECT
  e.id         AS employee_id,
```

```
  e.first_name,
  e.last_name,
  YEAR(o.order_date)  AS anio,
  MONTH(o.order_date) AS mes,
  COUNT(o.id)         AS total_pedidos
FROM employees e
JOIN orders o ON o.employee_id = e.id
GROUP BY e.id, e.first_name, e.last_name, YEAR(o.order_date),
MONTH(o.order_date);


-- Consultar
SELECT * FROM vw_empleado_pedidos_mensuales
ORDER BY anio, mes, employee_id;
```

OJO: tienes que agrupar por todas esa variables porque si agrupas solo por id, obtienes el total por empleado (sin desglose de fechas) y si agrupas por mes y año obtienes el nivel de detalle global mensual (todos los empleados juntos).

4. Crea una vista con órdenes de shipping_fee >= 100. Después modifícala para incluir taxes y una columna calculada total_extra.

```
-- Crear versión inicial
CREATE OR REPLACE VIEW vw_ordenes_envio_alto AS
SELECT       o.id,       o.customer_id,       o.ship_city,
o.ship_state_province, o.shipping_fee
FROM orders o
WHERE o.shipping_fee >= 100;
-- Modificar la vista (ALTER VIEW)
ALTER VIEW vw_ordenes_envio_alto AS
SELECT
```

```
  o.id, o.customer_id, o.ship_city, o.ship_state_province,
  o.shipping_fee, o.taxes,
  COALESCE(o.shipping_fee, 0) + COALESCE(o.taxes, 0) AS
total_extra
FROM orders o
WHERE o.shipping_fee >= 100;
```

5. Crea una vista con el número de productos por proveedor. Bórrala y luego recréala agregando el precio_promedio.

```
-- Crear vista
CREATE OR REPLACE VIEW vw_proveedores_productos AS
SELECT
  s.id AS supplier_id, s.company, s.state_province,
  COUNT(p.id) AS total_productos
FROM suppliers s
LEFT JOIN products p ON p.supplier_ids = s.id
GROUP BY s.id, s.company, s.state_province;
-- Borrar
DROP VIEW IF EXISTS vw_proveedores_productos;
-- Recrear con campo adicional
CREATE VIEW vw_proveedores_productos AS
SELECT
  s.id AS supplier_id, s.company, s.state_province,
  COUNT(p.id) AS total_productos,
  ROUND(AVG(p.list_price), 2) AS precio_promedio
FROM suppliers s
LEFT JOIN products p ON p.supplier_ids = s.id
GROUP BY s.id, s.company, s.state_province;
```

Video 10- Capítulo 9: Procesos Almacenados

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap9Ejercicios.sql".

1. Crea un procedimiento sp_ping() que regrese 'OK' y la fecha/hora actual.

```
DELIMITER //

CREATE PROCEDURE sp_ping()

BEGIN

  SELECT 'OK' AS status, NOW() AS ts;

END//

DELIMITER ;


-- Uso:

-- CALL sp_ping();
```

OJO: Los procedimientos almacenados son como funciones que guardas en la base de datos para poder reutilizarlas. En este caso no recibe parámetros (`()` vacío). (**`BEGIN ... END`**:) Delimita el bloque de instrucciones que ejecutará el procedimiento.

OJO: **`DELIMITER //`**. El delimitador por defecto de comandos es `;` pero como dentro de un `CREATE PROCEDURE` necesitas usar `;` para separar sentencias internas, cambias temporalmente el delimitador al símbolo `//` (podría ser cualquier otro: $$, @@, o ###). Así MySQL no corta el bloque antes de tiempo.

OJO: Puedes correrlo desde `Stored Procedures` en `Schemas o bien,` desde el Query pero sólo corriendo esa línea.

2. Crea sp_resumen_clientes() que liste company, total_ordenes, total_envio, total_impuestos, ultima_orden.

```
DELIMITER //

CREATE PROCEDURE sp_resumen_clientes()

BEGIN

  SELECT

    c.company,
```

```
    COUNT(o.id)                          AS total_ordenes,
    ROUND(SUM(COALESCE(o.shipping_fee,   0)),   2)   AS
total_envio,
    ROUND(SUM(COALESCE(o.taxes,  0)),  2)            AS
total_impuestos,
    MAX(o.order_date)                    AS ultima_orden
  FROM customers c
  JOIN orders o ON o.customer_id = c.id
  GROUP BY c.company
  ORDER BY total_ordenes DESC;
END//
DELIMITER ;


-- CALL sp_resumen_clientes();
```

3. Crea sp_pedidos_por_estado(p_state) (parámetro IN) que muestre #pedidos y total_envio por ciudad del estado dado (ship_state_province en orders).

```
DELIMITER //
CREATE   PROCEDURE   sp_pedidos_por_estado(IN   p_state
VARCHAR(50))
BEGIN
  SELECT o.ship_city,
         COUNT(*) AS total_pedidos,
         ROUND(SUM(COALESCE(o.shipping_fee,0)),   2)   AS
total_envio
  FROM orders o
  WHERE o.ship_state_province = p_state
  GROUP BY o.ship_city
  ORDER BY total_pedidos DESC;
```

```
END//

DELIMITER ;


-- CALL sp_pedidos_por_estado('NY');
```

OJO: Vamos a ver la convención de nombres de "**p_**" **(**parámetro**).** Se usa para variables que vienen como parámetros de entrada o salid**a**

OJO: `VARCHAR` = Variable Character, tipo de dato de texto variable en SQL y el número entre paréntesis (`50`) = longitud máxima permitida en caracteres.

OJO: `IN` (entrada): Valor que pasas al llamar el procedimiento. Solo sirve para *entrar datos*, no para devolverlos. En este caso recibirá el estado/provincia que quieras consultar.

4. Crea sp_pedidos_por_estado_default(p_state) que use 'NY' si el parámetro en ship_state_province viene NULL o vacío (default simulado).

```
DELIMITER //

CREATE PROCEDURE sp_pedidos_por_estado_default(IN p_state
VARCHAR(50))

BEGIN

  DECLARE v_state VARCHAR(50);

  SET v_state = COALESCE(NULLIF(p_state, ''), 'NY');


  SELECT o.ship_city,

         COUNT(*) AS total_pedidos,

         ROUND(SUM(COALESCE(o.shipping_fee,0)),    2)    AS
total_envio

  FROM orders o

  WHERE o.ship_state_province = v_state

  GROUP BY o.ship_city

  ORDER BY total_pedidos DESC;

END//
```

```
DELIMITER ;


-- CALL sp_pedidos_por_estado_default(NULL);
```

- **OJO**: Vamos a ver la convención de nombres de "**v_**" **(**variable local**).** Para variables declaradas dentro del procedimiento con `DECLARE`.

  `DECLARE v_state VARCHAR(50);` Crea una variable local llamada `v_state` que servirá para decidir qué estado usar realmente en la consulta.

  `SET v_state = COALESCE(NULLIF(p_state, ''), 'NY');`

  - `NULLIF(p_state, '')` → si `p_state` es una cadena vacía (`''`), lo convierte en `NULL`.
  - `COALESCE(..., 'NY')` → si el valor es `NULL`, usa `'NY'` como valor por defecto.

5. Crea sp_producto_info(p_product_id) que valide el id y lance SIGNAL si no existe; si existe, devuelva sus datos.

```
DELIMITER //

CREATE PROCEDURE sp_producto_info(IN p_product_id INT)

BEGIN

  IF NOT EXISTS (SELECT 1 FROM products WHERE id =
p_product_id) THEN

    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Producto no
encontrado';

  END IF;


  SELECT id, product_name, category, standard_cost,
list_price, supplier_ids

  FROM products

  WHERE id = p_product_id;

END//
```

```
DELIMITER ;

-- CALL sp_producto_info(1);
```

OJO: **`IN p_product_id INT`**: Parámetro de entrada: recibe un número (el ID del producto).

OJO: `SIGNAL` = instrucción para lanzar un **error manual** en SQL. `'45000'` es un código genérico de error definido por el usuario y `MESSAGE_TEXT = 'Producto no encontrado'` es el mensaje personalizado.

6. Crea sp_totales_cliente(p_customer_id, p_total_ordenes OUT, p_total_envio OUT) usando parámetros OUT.

```
DELIMITER //

CREATE PROCEDURE sp_totales_cliente(IN p_customer_id INT,

                                    OUT p_total_ordenes INT,

                                    OUT         p_total_envio
DECIMAL(12,2))

BEGIN

  SELECT                                        COUNT(o.id),
ROUND(SUM(COALESCE(o.shipping_fee,0)),2)

    INTO p_total_ordenes, p_total_envio

  FROM orders o

  WHERE o.customer_id = p_customer_id;

END//

DELIMITER ;

-- Primero defines variables de usuario para recibir los
resultados

SET @ord := 0;

SET @env := 0;


-- Llamas al procedimiento

CALL sp_totales_cliente(4, @ord, @env);
```

```
-- Ahora puedes leer las variables
SELECT @ord AS total_ordenes, @env AS total_envio;
```

OJO: `DECIMAL(12,2)` →hasta 12 dígitos en total, con 2 decimales.

OJO: `INTO p_total_ordenes, p_total_envio` → guarda esos resultados directamente en los parámetros de salida**.**

7. Crea sp_resumen_orden(p_order_id) que calcule subtotal_lineas, shipping_fee, taxes y total_general con variables locales.

```
DELIMITER //
CREATE PROCEDURE sp_resumen_orden(IN p_order_id INT)
BEGIN
  DECLARE v_linea_total DECIMAL(14,2);
  DECLARE v_envio DECIMAL(14,2);
  DECLARE v_impuestos DECIMAL(14,2);
  DECLARE v_total DECIMAL(14,2);

  SELECT COALESCE(SUM(od.quantity * od.unit_price * (1 -
COALESCE(od.discount,0))),0)
    INTO v_linea_total
  FROM order_details od
  WHERE od.order_id = p_order_id;

  SELECT COALESCE(o.shipping_fee,0), COALESCE(o.taxes,0)
    INTO v_envio, v_impuestos
  FROM orders o
  WHERE o.id = p_order_id;
```

```
  SET v_total = v_linea_total + v_envio + v_impuestos;

  SELECT p_order_id AS order_id,
         v_linea_total AS subtotal_lineas,
         v_envio AS shipping_fee,
         v_impuestos AS taxes,
         v_total AS total_general;
END//
DELIMITER ;

-- CALL sp_resumen_orden(30);
```

OJO: Empezamos declarando variables para guardar cálculos intermedios:

- `v_linea_total` → suma de los productos en la orden.
- `v_envio` → costo de envío.
- `v_impuestos` → impuestos.
- `v_total` → total final.

OJO: `SET v_total = v_linea_total + v_envio + v_impuestos;`
Suma subtotal + envío + impuestos.

8. Crea la función fn_total_linea(qty, unit_price, discount) que regrese el total de una línea (qty*unit_price*(1-discount)).

```
DELIMITER //
CREATE FUNCTION fn_total_linea(qty DECIMAL(12,3), pu
DECIMAL(10,2), d DECIMAL(6,4))
RETURNS DECIMAL(12,2) DETERMINISTIC
BEGIN
  RETURN ROUND(qty * pu * (1 - COALESCE(d,0)), 2);
END//
```

```
DELIMITER ;

-- SELECT id, order_id, fn_total_linea(quantity, unit_price,
discount) AS total_linea

-- FROM order_details;
```

OJO: Cuando creas una función en MySQL, debes indicar si es determinista **(**Los mismos valores de entrada, siempre devuelve el mismo resultado) o no determinista (Aunque le pases los mismos valores de entrada, puede devolver resultados distintos. Ejemplo: `fn_hoy()`**).**

OJO: Una función no se llama con CALL. Se usa dentro de un SELECT, WHERE, GROUP BY.

9. Crea la función fn_total_orden(p_order_id) que sume el total de todas las líneas de la orden (usa fn_total_linea).

```
DELIMITER //

CREATE FUNCTION fn_total_orden(p_order_id INT)

RETURNS DECIMAL(14,2) READS SQL DATA

BEGIN

  DECLARE total DECIMAL(14,2);

  SELECT  COALESCE(SUM(fn_total_linea(quantity,  unit_price,
discount)),0)

    INTO total

  FROM order_details

  WHERE order_id = p_order_id;

  RETURN total;

END//

DELIMITER ;


-- SELECT o.id, fn_total_orden(o.id) AS total_orden

-- FROM orders o;
```

OJO: `READS SQL DATA` indica que esta función **lee datos de tablas** (porque hace un `SELECT` dentro).

10. Crea sp_fill_suppliers_email() que complete emails faltantes en suppliers con un formato sugerido.

```
DELIMITER //

CREATE PROCEDURE sp_fill_suppliers_email()

BEGIN

  UPDATE suppliers s

  SET s.email_address = CONCAT('contact', s.id, '@',
LOWER(REPLACE(s.company, ' ', '')), '.com')

  WHERE s.email_address IS NULL OR s.email_address = '';

END//

DELIMITER ;


-- CALL sp_fill_suppliers_email();
```

Video 11- Capítulo 10.- Eventos Trigger

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap10Ejercicios.sql".

Un *trigger* es un objeto de base de datos asociado a una tabla que se activa automáticamente cuando ocurre un evento específico, como:

- `INSERT` → cuando se inserta una fila nueva
- `UPDATE` → cuando se modifica una fila existente
- `DELETE` → cuando se elimina una fila

Cuando defines un trigger, puedes indicar **cuándo** debe ejecutarse:

- `BEFORE` (antes de que se ejecute la operación en la tabla).
- `AFTER` (después de que la operación se haya completado).

Para visualizar los TRIGGERS es necesario utilizar el comando *SHOW TRIGGERS*.

1. Al insertar una orden, si 'taxes' viene NULL, calcularlo como ROUND(shipping_fee * tax_rate, 2), usando tax_rate si > 0; de lo contrario 0.16.

```
DROP TRIGGER IF EXISTS bi_orders_calc_taxes;

DELIMITER //

CREATE TRIGGER bi_orders_calc_taxes

BEFORE INSERT ON orders

FOR EACH ROW

BEGIN

  IF NEW.taxes IS NULL THEN

    SET NEW.taxes = ROUND(COALESCE(NEW.shipping_fee,0) *

                          COALESCE(NULLIF(NEW.tax_rate,0),
0.16), 2);

  END IF;

END//

DELIMITER ;
```

Desencadenar TRIGGER:

```
INSERT INTO orders (id, employee_id, customer_id, order_date,
shipping_fee, tax_rate, taxes)

VALUES (1001, 1, 1, NOW(), 200, 0.18, NULL);
```

- **OJO:**
  - Convención: `bi_` = *Before Insert*.
  - `NULLIF(a,b)` → devuelve `NULL` si `a=b`; si no, devuelve `a`.
  - `COALESCE(expr1, expr2, expr3, ...)`
    - Si `expr1` NO es NULL → devuelve `expr1`.
    - Si `expr1` es NULL → prueba con `expr2`.
  - `COALESCE(NULLIF(NEW.tax_rate,0), 0.16), 2)` aunque se pueda encontrar un poco redundante, `nullif` cumple el papel de tratar el valor **0** como si fuera **`NULL`.** Por ejemplo:
    - Si `tax_rate = 0.10` → `NULLIF(0.10,0)` = 0.10 → devuelve `0.10`.
    - Si `tax_rate = NULL` → `NULLIF(NULL,0)` = NULL → `COALESCE(...,0.16)` = 0.16.
    - Si `tax_rate = 0` → `NULLIF(0,0)` = NULL → `COALESCE(...,0.16)` = 0.16 (corrige el caso del cero).

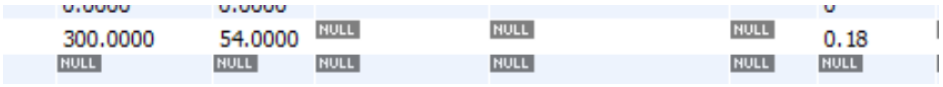
2. Al actualizar, si cambia shipping_fee o tax_rate, recalcular taxes con la misma regla.

```
DROP TRIGGER IF EXISTS bu_orders_recalc_taxes;

DELIMITER //

CREATE TRIGGER bu_orders_recalc_taxes

BEFORE UPDATE ON orders

FOR EACH ROW

BEGIN

  IF NOT (NEW.shipping_fee <=> OLD.shipping_fee)

     OR NOT (NEW.tax_rate    <=> OLD.tax_rate) THEN

    SET NEW.taxes = ROUND(COALESCE(NEW.shipping_fee,0) *

                          COALESCE(NULLIF(NEW.tax_rate,0),
0.16), 2);
```

```
  END IF;
END//
DELIMITER ;
```

Desencadenar TRIGGER:

```
UPDATE orders
SET shipping_fee = 300   -- cambió
WHERE id = 1001;
```



OJO:

- *Before Update* (`bu_`) : se ejecuta antes de que la fila actualizada se guarde.
- `OLD.columna` = valor antes del UPDATE (OLD: es palabra clave especial en SQL).
- `NEW.columna` = valor que estás intentando poner con el UPDATE (NEW: es palabra clave especial en SQL).

Entonces:

- `NEW.shipping_fee <=> OLD.shipping_fee` → compara ambos valores.
  - <=> es el operador de igualdad segura contra NULL.
  - A diferencia de =, este operador devuelve TRUE si ambos son `NULL`.
- `NOT (...)` → se cumple si el valor nuevo es diferente del viejo.

3. Evita inserciones con quantity <= 0, unit_price < 0 o discount fuera de [0,1). Si ocurre, lanzar SIGNAL con mensaje.

```
DROP TRIGGER IF EXISTS bi_order_details_validate;
DELIMITER //
CREATE TRIGGER bi_order_details_validate
BEFORE INSERT ON order_details
FOR EACH ROW
```

```
BEGIN
  IF NEW.quantity <= 0
     OR NEW.unit_price < 0
     OR COALESCE(NEW.discount,0) < 0
     OR COALESCE(NEW.discount,0) >= 1 THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'order_details inválido: quantity>0,
unit_price>=0, 0<=discount<1';
  END IF;
END//
DELIMITER ;
```

Desencadenar TRIGGER:

```
INSERT INTO order_details (id, order_id, product_id,
quantity, unit_price, discount)
VALUES (2001, 1001, 10, 0, 50, 0.10);
```

Error Code: 1644. order_details inválido: quantity>0, unit_price>=0, 0<=discount<1

4. Crear tabla de auditoría y registrar cambios de quantity, unit_price y discount tras UPDATE.

```
CREATE TABLE IF NOT EXISTS order_details_audit (
  audit_id        INT AUTO_INCREMENT PRIMARY KEY,
  order_detail_id INT,
  order_id        INT,
  product_id      INT,
  old_quantity    DECIMAL(12,3),
  new_quantity    DECIMAL(12,3),
```

```
  old_unit_price  DECIMAL(10,2),
  new_unit_price  DECIMAL(10,2),
  old_discount    DECIMAL(6,4),
  new_discount    DECIMAL(6,4),
  changed_at      DATETIME,
  changed_by      VARCHAR(255)
);

DROP TRIGGER IF EXISTS au_order_details_audit;
DELIMITER //
CREATE TRIGGER au_order_details_audit
AFTER UPDATE ON order_details
FOR EACH ROW
BEGIN
  IF (NOT (NEW.quantity   <=> OLD.quantity)) OR
     (NOT (NEW.unit_price <=> OLD.unit_price)) OR
     (NOT (NEW.discount   <=> OLD.discount)) THEN
    INSERT INTO order_details_audit
      (order_detail_id, order_id, product_id,
       old_quantity, new_quantity,
       old_unit_price, new_unit_price,
       old_discount, new_discount,
       changed_at, changed_by)
    VALUES
      (OLD.id, OLD.order_id, OLD.product_id,
       OLD.quantity, NEW.quantity,
       OLD.unit_price, NEW.unit_price,
```

```
        OLD.discount, NEW.discount,
        NOW(), CURRENT_USER());
    END IF;
END//
DELIMITER ;
```

Desencadenar TRIGGER:

```
UPDATE order_details
SET quantity = 20, unit_price = 45
WHERE id = 85;
SELECT * FROM northwind.order_details_audit;
```

audit_id	order_detail_id	order_id	product_id	old_quantity	new_quantity	old_unit_price	new_unit_price	old_discount	new_discount	changed_at	changed_by
1	85	58	52	40.000	20.000	7.00	45.00	0.0000	0.0000	2025-09-22 00:00:13	root@localhost
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

OJO:

- `AFTER UPDATE (au_)`: se ejecuta **después** de que MySQL guarda la actualización (ya no puedes modificar `NEW`, solo registrar).
- `FOR EACH ROW`: si un `UPDATE` afecta 10 filas, el trigger inserta 10 registros de auditoría (uno por fila).
- `INT:` Tipo de dato entero.
- `AUTO_INCREMENT:` cada vez que insertes una nueva fila, este campo se incrementa automáticamente en 1.
- `PRIMARY KEY:` define que esta columna será la clave primaria de la tabla.
- `DATETIME:` Tipo de dato `DATETIME` (En este caso, guarda una fecha y una hora completa)
- `VARCHAR(255):` Tipo de dato texto variable de hasta 255 caracteres (En este caso, representa el usuario que ejecutó el cambio en la base de datos**).**

OJO:

```
INSERT INTO nombre_tabla (columna1, columna2, columna3, ...)
VALUES (valor1, valor2, valor3, ...);
```

- `nombre_tabla:` la tabla donde quieres insertar datos.

- `(columna1, columna2, ...)`: lista de columnas a las que quieres asignar valores.
- `VALUES`: palabra clave que introduce los valores que se insertarán.
- `(valor1, valor2, ...)`: los valores correspondientes a cada columna.

5. Registrar en la misma tabla de auditoría cuando se borre una línea.

```
DROP TRIGGER IF EXISTS ad_order_details_audit_delete;

DELIMITER //

CREATE TRIGGER ad_order_details_audit_delete

AFTER DELETE ON order_details

FOR EACH ROW

BEGIN

  INSERT INTO order_details_audit

    (order_detail_id, order_id, product_id,

     old_quantity, new_quantity,

     old_unit_price, new_unit_price,

     old_discount, new_discount,

     changed_at, changed_by)

  VALUES

    (OLD.id, OLD.order_id, OLD.product_id,

     OLD.quantity, NULL,

     OLD.unit_price, NULL,

     OLD.discount, NULL,

     NOW(), CURRENT_USER());

END//

DELIMITER ;
```

Desencadenar TRIGGER:

```
DELETE FROM order_details
WHERE id = 91;

SELECT * FROM northwind.order_details_audit;
```

audit_id	order_detail_id	order_id	product_id	old_quantity	new_quantity	old_unit_price	new_unit_price	old_discount	new_discount	changed_at	changed_by
1	85	58	52	40.000	20.000	7.00	45.00	0.0000	0.0000	2025-09-22 00:00:13	root@localhost
2	91	81	56	0.000	NULL	38.00	NULL	0.0000	NULL	2025-09-22 00:06:36	root@localhost
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

- **OJO:** Tu trigger anterior es `AFTER UPDATE`, solo se dispara cuando se actualiza una fila en `order_details` y tu trigger nuevo es `AFTER DELETE` solo se dispara cuando se elimina una fila.

  `OLD.quantity`, `OLD.unit_price`, `OLD.discount` representan los valores que tenía la fila antes de borrarse.

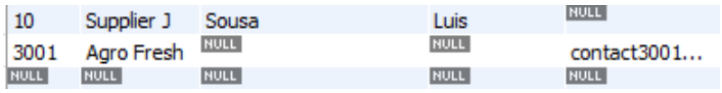
  `NULL` en `new_quantity`, `new_unit_price`, `new_discount` significa que después del DELETE **no hay valores nuevos**, porque la fila ya no existe en `order_details`.

6. Si email_address viene vacío/NULL, generar contact<ID>@<company_sinespacios>.com

```
DROP TRIGGER IF EXISTS bi_suppliers_fill_email;
DELIMITER //
CREATE TRIGGER bi_suppliers_fill_email
BEFORE INSERT ON suppliers
FOR EACH ROW
BEGIN
  IF NEW.email_address IS NULL OR NEW.email_address = '' THEN
    SET NEW.email_address = CONCAT('contact', NEW.id, '@',

LOWER(REPLACE(NEW.company, ' ', '')), '.com');
  END IF;
END//
DELIMITER ;
```

Desencadenar TRIGGER:

```
INSERT INTO suppliers (id, company, email_address)
VALUES (3001, 'Agro Fresh', NULL);
```



OJO: El trigger se ejecuta antes de que se inserte una fila en la tabla `suppliers`. Es decir, tiene la oportunidad de modificar los valores que se insertarán.

7. Genera un evento que garantice que cada día se rellenen automáticamente los impuestos (**`taxes`**) faltantes en `orders`, calculándolos a partir de la tarifa de envío (`shipping_fee`) y la tasa de impuestos (`tax_rate`), usando un valor por defecto del 16% cuando no hay tasa válida.

```
DROP EVENT IF EXISTS ev_fill_missing_taxes;
DELIMITER //
CREATE EVENT ev_fill_missing_taxes
ON SCHEDULE EVERY 1 DAY
DO
  UPDATE orders
  SET taxes = ROUND(COALESCE(shipping_fee,0) *
COALESCE(NULLIF(tax_rate,0), 0.16), 2)
  WHERE taxes IS NULL;
//
DELIMITER ;
```

Video 12- Capítulo 14: Indexing

NOTA: Las queries de respuesta podrán ser consultadas adicionalmente en su archivo .sql titulado "cap14Ejercicios.sql".

1. Crear índice en orders(order_date) y probar consulta por rango.

```
-- Prueba previa
EXPLAIN SELECT id, customer_id, order_date
FROM orders
WHERE order_date BETWEEN '2006-01-01' AND '2006-03-31';


-- Crear índice
CREATE INDEX idx_orders_order_date ON orders (order_date);


-- Prueba posterior
EXPLAIN SELECT id, customer_id, order_date
FROM orders
WHERE order_date BETWEEN '2006-01-01' AND '2006-03-31';
```

Primera prueba (sin índice)

- **type = ALL**
  Significa Full Table Scan**:** MySQL recorre toda la tabla **`orders`** para verificar si cada fila cumple la condición de fecha. Esto es lo menos eficiente.
- **possible_keys = NULL**
  No había ningún índice que pudiera ayudar.
- **rows = 48**: MySQL estima que revisará las 48 filas de la tabla.
- **filtered = 11.11**: De todas las filas revisadas, solo espera que ~11% cumpla la condición.
- **Extra = Using where**: El filtrado se hace directamente con la cláusula `WHERE`, sin ayuda de índice.

En resumen: se escanea toda la tabla aunque solo te interesen unas cuantas filas.

Segunda prueba (con índice en order_date)

- **type = range**: Ahora MySQL usa un **índice por rango**, que es mucho más eficiente. Va directo al bloque de fechas que le interesan, en lugar de revisar toda la tabla.
- **possible_keys = idx_orders_order_date**: Muestra que el índice recién creado es candidato para esta consulta.
- **key = idx_orders_order_date**: Confirma que ese índice está siendo usado.
- **rows = 15**: Ahora MySQL estima que solo necesitará revisar 15 filas (muchísimo menos que las 48 de antes).
- **filtered = 100.00**: El índice permite acceder exactamente a las filas necesarias, sin desperdicio.
- **Extra = Using index condition**: Significa que está aplicando la condición del `WHERE` directamente dentro del índice, evitando tener que ir a leer todas las filas completas de la tabla.

  En resumen: ahora MySQL usa el índice y la consulta es mucho más rápida**.**

Una vez creado el índice los tiempos de consulta disminuyen a 0:

Duration / Fetch
0.015 sec / 0.000 sec
0.062 sec
0.000 sec / 0.000 sec

2. Crear índice en orders(customer_id) y consultar órdenes de un cliente.

```
-- Prueba previa
EXPLAIN
SELECT id, order_date, shipping_fee
FROM orders
WHERE customer_id = 4
ORDER BY order_date DESC;

-- Crear índice
CREATE INDEX idx_orders_customer ON orders (customer_id);

-- Prueba posterior
EXPLAIN
SELECT id, order_date, shipping_fee
FROM orders
```

```
WHERE customer_id = 4
ORDER BY order_date DESC;

-- En la base de datos
SELECT id, order_date, shipping_fee
FROM orders
WHERE customer_id = 4
ORDER BY order_date DESC;
```

3. Listar índices de orders y eliminar idx_orders_order_date.

```
-- Mostrar
SHOW INDEX FROM orders;
-- Eliminar (si el nombre coincide)
DROP INDEX idx_orders_order_date ON orders;
```

4. Crear índice de prefijo para acelerar búsquedas por company en suppliers.

```
CREATE INDEX idx_suppliers_company_pref ON suppliers
(company(20));

SELECT id, company, state_province
FROM suppliers
WHERE company LIKE 'Supplier%';
```

- **OJO**: `company(20))`: Indexa solo los primeros 20 caracteres de la columna.

  `LIKE 'Supplier%'`: Cualquier texto que empiece con la palabra `Supplier` y luego tenga cualquier cosa o incluso nada.

5. Crear FULLTEXT sobre products(product_name, description) y buscar por texto.

```
CREATE FULLTEXT INDEX ftx_products_name_desc
ON products (product_name, description);
```

```
-- Búsqueda en modo natural

SELECT id, product_name

FROM products

WHERE MATCH(product_name, description) AGAINST ('avocado' IN
NATURAL LANGUAGE MODE);
```

- **OJO**: Un FULLTEXT INDEX es una estructura especial que la base de datos crea para guardar cada palabra que aparece en tus columnas de texto**,** junto con en qué filas aparece**.** Entonces cuando haces una búsqueda FULLTEXT, MySQL no escanea todos los textos. Va directo a este "diccionario invertido" y devuelve las filas relevantes.

  La parte `MATCH(...)` le dice a MySQL:"Quiero hacer una **búsqueda de texto** en estas columnas."

  - `product_name` y `description` deben tener un índice FULLTEXT (o formar parte de uno). `MATCH` no funciona si no has creado previamente un índice FULLTEXT.

6. Crear índice en order_details(order_id, product_id) y consultarlo.

```
CREATE INDEX idx_od_order_product ON order_details (order_id,
product_id);

SELECT id, quantity, unit_price

FROM order_details

WHERE order_id = 30

  AND product_id IN (7, 51, 80);
```

OJO: Este índice cubre dos columnas (índice compuesto)**:** primero `order_id` y luego `product_id` y el orden importa**:**

- Se puede usar para búsquedas por `order_id`.
- Se puede usar para búsquedas por `order_id` **y** `product_id`.
- No se aprovecha bien si buscas solo por `product_id` (porque `order_id` es la primera columna).

Piensa en este índice como un libro ordenado primero por **`order_id`** y, dentro de cada `order_id`, por `product_id`.

7. Crear orders(customer_id, order_date) y probar 3 consultas.

```
CREATE INDEX idx_orders_customer_date ON orders (customer_id,
order_date);

-- (a) Usa ambas columnas

SELECT id, order_date

FROM orders

WHERE customer_id = 4

  AND order_date >= '2006-01-01'

ORDER BY order_date;


-- (b) Solo la 1ª columna

SELECT id, order_date

FROM orders

WHERE customer_id = 4;


-- (c) Solo la 2ª columna (no suele usar el índice compuesto)

SELECT id, customer_id

FROM orders

WHERE order_date >= '2006-01-01';
```

OJO: En índices compuestos, el orden importa**:**

- El índice se puede usar si filtras por la primera columna (`customer_id`), o por las dos columnas (`customer_id` + `order_date`) pero no se aprovecha bien si filtras solo por la segunda columna (`order_date`).

OJO: Caso (a): Usa ambas columnas

Aquí MySQL aprovecha **todo el índice**:

- Encuentra rápidamente las filas con `customer_id = 4`.
- Dentro de ese rango, usa también `order_date` para filtrar y ordenar.
- Además, como el índice ya está ordenado por `(customer_id, order_date)`, evita hacer un *filesort*.

OJO: Caso (b): Solo la 1ª columna

Aquí MySQL usa el índice, pero **solo la parte de `customer_id`**:

- Encuentra todas las filas de ese cliente.
- No puede aprovechar `order_date` porque no hay filtro ni ordenamiento sobre esa columna.

OJO: Caso (c): Solo la 2ª columna

Aquí MySQL **no suele usar el índice compuesto** porque el índice empieza por `customer_id`.

- Como no filtras por `customer_id`, el índice no ayuda a restringir filas.
- Puede que MySQL haga un **full table scan** (recorre toda la tabla).
- Excepción: si hubiera otro índice sobre `order_date` por separado, usaría ese.

8. Mostrar IGNORE INDEX y FORCE INDEX en orders.

```
-- Ignorar
SELECT id, customer_id, order_date
FROM orders IGNORE INDEX (idx_orders_customer_date)
WHERE customer_id = 4
  AND order_date >= '2006-01-01';


-- Forzar
SELECT id, customer_id, order_date
FROM orders FORCE INDEX (idx_orders_customer_date)
WHERE customer_id = 4
  AND order_date >= '2006-01-01';
```